UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

20|21

Trabajo fin de grado

Portal web para el análisis estático y dinámico de aplicaciones Android

Arturo Blázquez Pérez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Portal web para el análisis estático y dinámico de aplicaciones Android

Autor: Arturo Blázquez Pérez
Tutor: Carlos Aguirre Maeso

junio 2021

Arturo Blázquez Pérez
Portal web para el análisis estático y dinámico de aplicaciones Android

Arturo Blázquez Pérez
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y amigos

Los que pueden imaginar cualquier cosa, pueden crear lo imposible.

Alan Turing

Agradecimientos

Quiero agradecer a mi familia por el apoyo que me han dado durante todos estos años y por tener la paciencia de aguantarme durante todo este largo viaje.

Quiero agradecer también a mis amigos porque han estado siempre presentes para ayudarme y su apoyo me ha ayudado en los momentos difíciles.

Por último me gustaría agradecer a Alejandro Martín García por haber dirigido este trabajo y ayudarme con todas las dudas que me han ido surgiendo.

Resumen

Uno de los ciberataques más frecuentes en los últimos años es la instalación de malware en dispositivos Android. Por ello, es fundamental estudiar nuevas formas de detectar este tipo de amenazas. Uno de los métodos más efectivos es usando *Machine Learning*.

Este TFG consiste en el desarrollo de un portal web que facilita la generación de análisis estáticos y dinámicos de aplicaciones Android usando la herramienta AndroPyTool. Los usuarios del portal web podrán tanto analizar nuevas aplicaciones como consultar y descargar informes detallados de aplicaciones ya analizadas. Este portal permitirá además la recopilación de datos de nuevas aplicaciones maliciosas con el objetivo de entrenar modelos de *Machine Learning* para su detección.

El uso del portal es tan sencillo como arrastrar y soltar una aplicación Android que se quiera analizar. El portal automáticamente analiza la aplicación y nos muestra un informe detallado que podemos visualizar y descargar. Internamente, se guarda un registro de todas las aplicaciones analizadas, permitiendo compartir el análisis de cualquiera de ellas de una manera rápida y sencilla, facilitando así el trabajo de los analistas de *malware*. Además, el portal web está enfocado en ofrecer una experiencia de usuario muy intuitiva, permitiendo el acceso desde cualquier tipo de dispositivo ya sea móvil, ordenador o tableta.

Para la creación de este portal web ha sido necesario el desarrollo de dos proyectos, uno *back end* y otro *front end*. El primero expone una API REST que, usando Python y Flask, permite usar la herramienta AndroPyTool como un servicio web sin necesidad de abrir una terminal. El otro usa Angular y la librería de componentes Material Design para ofrecer un portal web que consume esa API y facilita el uso de la herramienta.

Palabras clave

Android, malware, AndroPyTool, Python, Flask, Angular, Docker, Material Design, API, backend, frontend, DevOps, Arquitectura hexagonal, TypeScript

Abstract

Malware applications targeting the Android platform are among the most used tactics nowadays to perpetrate a cyberattack, so it is critical to study new ways to detect them. One of the most effective methods to detect malware is using Machine Learning.

This work consists on the development of a web portal that helps in the creation of dynamic and static features from Android APKs using AndroPyTool. Users of this web are able to analyze new applications as well as viewing and downloading reports of applications already analyzed. This web will also allow the compilation of malware's data, helping the training of Machine Learning models that detect malware.

The use of the web portal is as easy as dragging and dropping an Android application that you wish to analyze. The web automatically scans the application and shows a report that you can see and download. The web stores a registry of all analyzed applications, allowing an easy sharing of the report of any application. This will help the work of malware analysts. The web portal also offers a very intuitive user experience, allowing the use of the application from mobile, tablet or computer.

For the creation of this web portal it has been needed the creation of two projects, a back-end and a front-end one. The back-end uses Python and Flask to expose an API REST that allows the use of the tool without opening a terminal. The front-end uses Angular and Material Design component library to show a web that consumes this API and helps with the use of the tool.

Keywords

Android, malware, AndroPyTool, Python, Flask, Angular, Docker, Material Design, API, backend, frontend, DevOps, Hexagonal Architecture, TypeScript

Índice

Listas

Lista de figuras

1

Introducción

Debido a la popularidad de los smartphones y las tabletas, una de las herramientas más utilizadas para perpetrar un ciberataque son las aplicaciones *malware* para dispositivos Android. Es por ello que es fundamental estudiar nuevas formas de detectarlas. Uno de los métodos más efectivos para diferenciar *malware* y *benignware* es usando técnicas de *machine learning*. Por desgracia, para entrenar ese tipo de clasificadores se necesita de una gran cantidad de datos etiquetados [1].

Bajo esta premisa se creó AndroPyTool [1, 2], una herramienta que permite extraer características estáticas y dinámicas de aplicaciones de Android. Permite a los analistas de *malware* integrar en un entorno único los resultados de diferentes herramientas de análisis de aplicaciones de Android como DroidBox, FlowDroid, Strace, AndroGuard o VirusTotal.

El proyecto actual pretende extender y facilitar el uso de la herramienta AndroPyTool, mediante la creación de un portal web que permite usar la herramienta sin necesidad de instalar nada.

1.1. Motivación

La manera más sencilla de ejecutar AndroPyTool es usando Docker. Lo único que hay que hacer es bajarse la imagen que está subida en Docker Hub y correrla. El principal inconveniente de este modo de usar la herramienta es que el análisis de aplicaciones es un proceso muy costoso y al ejecutarlo en un contenedor sólo podemos usar una cantidad limitada de los recursos de nuestra máquina. Esto es especialmente relevante a la hora de hacer el análisis dinámico, ya que necesitamos asignar una gran cantidad de RAM para el proceso o sino el resultado final no será tan fiable.

La segunda manera de utilizar la herramienta es instalarla directamente en nuestra máquina. La principal desventaja de este método es que tenemos que instalar todas las dependencias del proyecto a mano, ejecutando múltiples comandos en la terminal. Este procedimiento requiere de mucho tiempo, pudiendo llegar a ser horas para quien no esté habituado a los comandos que se usan. Además este proceso sólo ha sido probado en Ubuntu, haciendo prácticamente imposible usarlo en otros sistemas operativos como MacOS o Windows.

Que estas sean las únicas maneras de usar la herramienta, implica que los posibles usuarios de AndroPyTool necesitan de un alto conocimiento usando la terminal y disponer de una máquina muy potente. Este es un subconjunto muy pequeño de los posibles usuarios que podrían utilizar la herramienta.

Ante esta situación se plantea la creación de un portal web que permita la utilización de la herramienta AndroPyTool sin necesidad de instalarla en nuestro equipo. Esto proporciona una experiencia de usuario muy sencilla, ya que para utilizarla lo único que se necesita hacer es subir un archivo a una página web. De esta manera solucionamos todos los problemas mencionados previamente y, al ser el portal web el que se encarga de analizar la aplicación, no necesitamos de una máquina potente para realizar el análisis, pudiendo hacerlo incluso desde un móvil.

1.2. Objetivos

El objetivo de este proyecto es la creación de un portal web que permita el análisis de aplicaciones Android. El portal permitirá también la visualización y descarga de los análisis generados.

El proyecto tiene 3 partes fundamentales: *back end*, *front end* y *DevOps*.

- La parte de ***backend*** ha consistido fundamentalmente en adaptar AndroPyTool para permitir usarlo mediante una API REST en vez de por terminal. La API tiene 2 *endpoints* bien diferenciados, uno para subir aplicaciones y otro para ver los informes generados.
- La parte de ***front end*** se ha centrado en la experiencia de usuario, haciendo transparente al usuario el funcionamiento interno del portal. La página principal de la web consiste en una caja donde puedes tanto subir una aplicación para analizarla como buscar aplicaciones ya analizadas. La segunda página más importante presenta el informe del análisis de una aplicación, permitiendo navegar cómodamente por los diferentes tipos de análisis y descargarlos de manera individual o conjunta.
- La parte de ***DevOps*** tiene dos ejes fundamentales. Primero, un conjunto de buenas prácticas para permitir un desarrollo más rápido y de mayor calidad. Segundo, una separación del desarrollo de la aplicación con su despliegue, permitiendo así tener un mejor control de los ciclos de vida de las aplicaciones desarrolladas.

1.3. Estructura de la memoria

En esta primera parte de la memoria hemos hecho una introducción del proyecto, hablando de la motivación y objetivos del mismo. En la segunda parte explicaremos las tecnologías usadas en cada parte del proyecto. En la tercera parte hablaremos del diseño que se ha seguido con cada una de las

partes del proyecto, explicando las decisiones que se han tomado y por qué. Hablaremos también del desarrollo que se ha llevado a cabo siguiendo esos diseños. Finalmente, en el capítulo 4 hablaremos de las conclusiones del proyecto y trabajo futuro que se puede llevar a cabo.

2

Estado del arte

Los últimos informes sobre aplicaciones maliciosas evidencian un serio problema causado por el *malware* para móviles [3]. Este tipo de programas es cada vez más inteligente y afecta a más usuarios. La constante aparición de nuevos tipos de virus y otros tipos de aplicaciones maliciosas ha supuesto un importante aumento de estudios sobre el tema, siendo Android el principal foco de muchos de estos estudios, debido a la gran cantidad de ataques dirigidos hacia esta plataforma. Muchos de estos trabajos se centran en la construcción de herramientas para identificar si una aplicación es *benignware* o *malware*.

Este tipo de herramientas normalmente se basa en técnicas de aprendizaje automático, para lo cual primero es necesario analizar y extraer una serie de características de aplicaciones para Android. Una vez se tiene un conjunto de datos suficientemente grande se puede entrenar un clasificador que diferencie en *malware* y *benignware*.

Se han creado multitud de herramientas que analizan aplicaciones Android y extraen sus características. Estas herramientas normalmente se enfocan en realizar uno de los siguientes 3 tipos de análisis:

- Análisis pre estático: extrae meta información del fichero *apk* sin analizar el código de esta.
- Análisis estático: Analiza el código de la aplicación y extrae características como llamadas a la API, permisos requeridos por la aplicación o llamadas al sistema, entre otras.
- Análisis dinámico: Ejecuta la aplicación en un entorno controlado y controla qué acciones realiza, consiguiendo así datos sobre el comportamiento real de una aplicación.

Dentro de las herramientas que realizan análisis estático destaca Androguard [4], diseñada para extraer características de los archivos del paquete de una aplicación. DroidBox [5] es una herramienta muy popular que levanta un emulador de Android, ejecuta la aplicación a analizar dentro de ese emulador y supervisa cada una de las acciones realizadas por la aplicación dentro de este emulador. Existen múltiples otras herramientas para extraer este tipo de datos, como pueden ser Apktool [6], Dex2jar [7] o FlowDroid [8]. Por desgracia, si se quiere un análisis tanto estático como dinámico, necesitaremos usar múltiples herramientas diferentes para conseguir estos datos.

Por esta razón se crea AndroPyTool [1, 2], una herramienta que integra múltiples herramientas de análisis de aplicaciones Android y genera un informe con datos pre estáticos, estáticos y dinámicos de las aplicaciones analizadas. El uso de esta herramienta permite a los analistas de *malware* obtener análisis completos de manera sencilla y rápida.

2.1. AndroPyTool

AndroPyTool es una herramienta desarrollado en Python que integra los resultados de diferentes análisis de aplicaciones Android en un solo informe completo y detallado. Estos análisis son tanto estáticos como dinámicos.

La herramienta realiza 7 pasos para analizar aplicaciones:

- **Paso 1: Filtrado de Apks.** Se utiliza Androguard para comprobar si el fichero analizado es una aplicación Android válida.
- **Paso 2: Análisis de VirusTotal.** Se utiliza la API de VirusTotal para generar un informe de más de 60 antivirus.
- **Paso 3: Particionado del *dataset*.** Se utilizan los resultados de VirusTotal para dividir las aplicaciones analizadas en *malware* y *benignware*.
- **Paso 4: Ejecución de FlowDroid.** Se obtiene el análisis estático de FlowDroid.
- **Paso 5: Procesamiento de la salida de FlowDroid.** Se procesan los archivos generados por FlowDroid.
- **Paso 6: Ejecución de DroidBox.** Se ejecuta un análisis dinámico usando DroidBox. La versión de DroidBox usada incluye algunas modificaciones como la integración con Strace.
- **Paso 7: Extracción de características y procesado.** Se extraen el resto de características y se procesan los diferentes análisis para integrarlos en uno solo.

2.2. Tecnologías empleadas

Cada parte del proyecto ha utilizado tecnologías, lenguajes de programación y librerías completamente diferentes. En las siguientes subsecciones lo veremos más en detalle.

2.2.1. Back end

Hemos elegido Python como lenguaje para desarrollar el *back end* de nuestro proyecto, ya que AndroPyTool también ha sido desarrollada usando Python. De esta manera nos aseguramos una com-

patibilidad al 100 % y una mayor velocidad de desarrollo.

Para exponer la API REST hemos usado un *framework* para aplicaciones web, que es un entorno de trabajo diseñado para facilitar la tarea de creación de servicios web, permitiendo la utilización de bibliotecas estándar y que facilitan la reutilización de código.

Existen múltiples *frameworks* en Python, siendo el más conocido Django. Para este proyecto, en el que sólo vamos a exponer una API y no vamos a servir directamente páginas web, Django es un *framework* demasiado pesado y con demasiada funcionalidad que no va a ser usada. Hemos decidido usar lo que se conoce como un micro web *framework*, que es un *framework* que no necesita de la utilización de librerías o herramientas específicas.

De este tipo de micro *frameworks* se eligieron 2, Flask Y Falcon, cada uno con sus respectivas ventajas y desventajas. Falcon es más nuevo y prometedor, ya que está enfocado únicamente en la creación de APIs, sin incluir lógica relacionado con servir páginas web. Esto le permite ser hasta 10 veces más rápido que Django y hasta 3 veces más rápido que Flask. Por su parte, Flask es el *framework* más asentado y con una mejor documentación.

Tras realizar varias pruebas de concepto, se llegó a la conclusión de que la falta de documentación de Falcon iba a suponer unos tiempos de desarrollo muy superiores con respecto a usar Flask. Además esta falta de documentación implica tener una menor seguridad de la calidad del código generado. Como se prevé que la cantidad de peticiones que se van a realizar no es demasiado grande y que el principal cuello de botella es el análisis de las aplicaciones, se ha decidido usar Flask [10].

2.2.2. Front end

Para desarrollar parte de *front end* hemos elegido Angular. Hemos usado la versión 10.2, que es la versión estable más nueva que había en el comienzo del desarrollo del proyecto. Angular es un *framework* para el desarrollo de páginas SPA (*single-page application*), que son páginas dinámicas en las que ante la interacción del usuario se reescribe la página con la información enviada desde un servidor web en vez de cargar una página nueva desde cero. Esto hace que las transiciones dentro de la página sean más fluidas, siendo el rendimiento comparable con el de una app nativa.

Como lenguajes de programación hemos usado html, scss y TypeScript. Scss es una extensión de css (todo código css válido es un código scss válido) que permite, entre otras, el uso de variables o anidar código. TypeScript es un lenguaje que extiende a JavaScript, cuya principal cualidad es que permite declarar los tipos de las variables, creando así un código con menos probabilidad de errores y con mejores ayudas a la hora de usar un IDE.

Angular nos permite usar una librería de componentes, que es básicamente una colección de interfaces de usuario como botones, desplegables o barras, entre otros. Hemos decidido usar la librería

de Material Design, debido a la gran cantidad de componentes que tiene, su flexibilidad y que es una librería muy bien documentada.

Hemos usado una gran cantidad de librerías, de entre las que destacan:

- **ngx-dropzone**: Es un componente que dibuja una caja en la que puedes arrastrar archivos para subirlos.
- **rxjs**: Nos permite hacer llamadas asíncronas, utilizando una abstracción, que son los *Observables*.
- **ngx-translate**: Nos permite crear una web en múltiples idiomas usando un fichero de traducción.
- **json-tree**: Es un componente que permite mostrar datos en formato json con una estructura de árbol.

2.2.3. DevOps

Hemos usado GitHub como repositorio de código y control de versiones. Existen otras plataformas para alojar proyectos usando Git, como GitLab, que ofrecen un mejor CI/CD (*Continuous Integration and Continuous Delivery*). Como el proyecto original estaba alojado en GitHub y al ser un proyecto de una sola persona no es tan necesaria la integración continua hemos decidido alojar los 2 proyectos (*front end* y *back end*) en GitHub. La parte del *back end* se ha creado con una rama del proyecto principal de AndroPyTool y para la parte del *front end* se ha creado un proyecto nuevo llamado AndroPyTool-angular.

Además durante el desarrollo del proyecto también se ha hecho uso de otras herramientas:

- **Desarrollo:** Para el desarrollo del proyecto se han usado los IDEs Pycharm para el *back end* y WebStorm para el *front end*. Estos entornos de desarrollo permiten, entre otras cosas, crear proyectos Flask y Angular, respectivamente, con la mayoría de ajustes predeterminados ya creados. Estos IDEs además hacen un análisis del código, encontrando automáticamente posibles fallos y corregirlos de un solo click, haciendo así que el desarrollo sea más rápido y sencillo y que el código generado sea de mayor calidad.
- **Control de versiones:** Para el control de versiones hemos usado GitKraken, que es un cliente gráfico de Git, muy intuitivo y que agiliza mucho el uso de todas las funcionalidades de Git.
- **Validación:** Para comprobar el correcto funcionamiento de la API hemos usado Postman, que es una herramienta que ayuda en el diseño y testing de APIs, simplificando de gran manera el proceso de creación de una API.
- **Despliegue:** Para el despliegue de las aplicaciones hemos usado un servidor proporcio-

nado por la Autónoma. Hemos usado un puerto para desplegar el *back end* y otro diferente para el *front end*. Hemos usado Docker, que nos permite levantar contenedores con todo lo necesario para servir nuestras aplicaciones sin necesidad de instalar nada directamente en el propio servidor, sino que se instala en una máquina virtual.

- **Back end:** Para el *back end* levantamos un contenedor de Ubuntu con Gunicorn, que es un servidor HTTP WSGI para Python. Para el *front end* levantamos un contenedor de Node y servimos la aplicaición con Apache2.

3

Diseño y desarrollo

En este apartado veremos las decisiones de diseño que se han tomado y por qué y la implementación que se ha hecho de estos diseños. Empezaremos con la parte de *back end* del proyecto, continuaremos con el *front end* y la integración de ambos y acabaremos con un apartado de *DevOps*.

3.1. Back end

3.1.1. Diseño

Para el *back end* hemos decidido usar una arquitectura hexagonal, cuya principal característica es la separación de responsabilidades en diferentes capas. De esta manera se consigue que las diferentes lógicas de la aplicación no estén acopladas, permitiendo un desarrollo y *testing* de cada una de ellas de manera independiente. Gracias a esto ganamos un mejor mantenimiento y estabilidad, ya que estamos separando todos los casos de uso y el modelo de dominio de la infraestructura.

La arquitectura hexagonal divide las responsabilidades en 3 capas bien diferenciadas: la interfaz de usuario, la lógica de negocio y la parte del servidor. Cada una de estas capas tiene definida una interfaz para contactar con ella, a la que se le llama puerto. Cada una de las implementaciones de estos puertos se llama adaptador. Podemos ver un diagrama de esta arquitectura en la figura 3.1.

La capa de la izquierda define cómo un usuario o una aplicación externa interactúa con nuestro programa. Esta capa puede ser una interfaz gráfica, una API o acceso por terminal, entre otros. En nuestro caso esta capa será la API REST y toda su lógica relacionada.

La capa central es la lógica de negocio. En nuestro caso se trata de la propia herramienta AndroPyTool.

La capa de la derecha incluye todo lo necesario para que nuestra aplicación funcione. Por ejemplo, el acceso a base de datos, llamadas al sistema o comunicación con otros programas. En nuestro caso esta capa es la que se encarga del almacenamiento de los informes y su consulta.

Una gran ventaja de esta arquitectura es que en cualquier momento podemos hacer cambios en

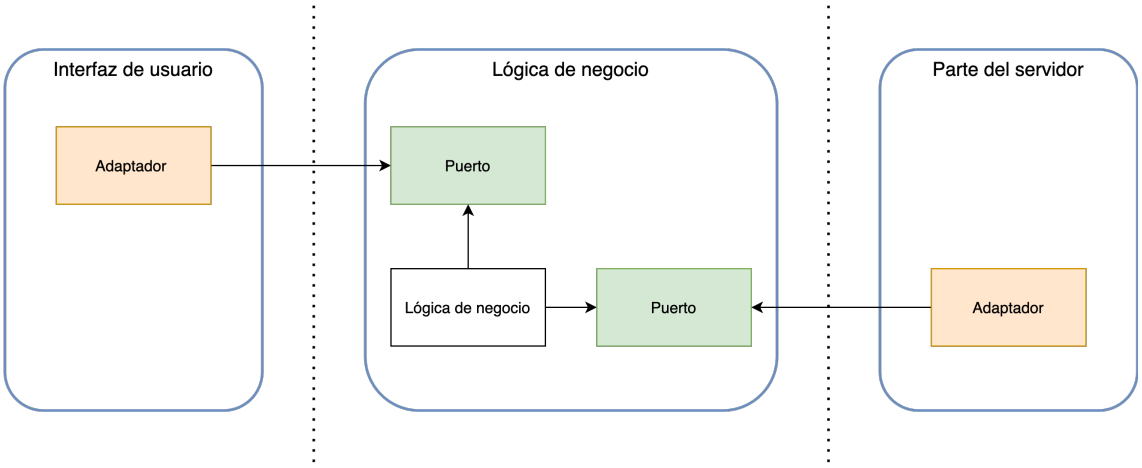


Figura 3.1: Diagrama de la arquitectura hexagonal.

cualquiera de las capas y las demás no se verían afectadas ya que no hemos cambiado la interfaz con la que se conectan entre ellas. Podríamos hasta cambiar una capa por una completamente nueva y no habría que hacer ningún cambio.

La lógica de negocio implementa la siguiente interfaz:

- **upload_apk(uploaded_file, virus_total_api_key)**: Analiza el fichero y devuelve su resource_uri.
- **get_all_reports()**: Devuelve todos los informes generados.
- **get_pre_static_analysis(sha256)**: Devuelve el análisis pre estático de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_dynamic_analysis(sha256)**: Devuelve el análisis dinámico de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_dynamic_analysis_droidbox(sha256)**: Devuelve sólo la parte de DroidBox del análisis dinámico de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_dynamic_analysis_strace(sha256)**: Devuelve sólo la parte de Strace del análisis dinámico de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_virusTotal_analysis(sha256)**: Devuelve el análisis de VirusTotal de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_static_analysis(sha256)**: Devuelve el análisis estático de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_static_analysis_androPyTool(sha256)**: Devuelve sólo la parte de AndroPyTool del análisis estático de la aplicación con ese sha256. En caso de que no exista devuelve un 404.
- **get_static_analysis_flowDroid(sha256)**: Devuelve sólo la parte de FlowDroid del análisis

estático de la aplicación con ese sha256. En caso de que no exista deveulve un 404.

- **get_complete_analysis(sha256)**: Devuelve el análisis completo de la aplicación con ese sha256. En caso de que no exista devuelve un 404.

La parte del servidor implementa la siguiente interfaz:

- **save_apk(sha256, uploaded_file)**: Guarda la apk.
- **app_has_report(sha256)**: Devuelve si la aplicación con ese sha256 ya ha sido analizada previamente.
- **save_report_to_db(sha256)**: Guarda el informe de la aplicación con ese sha256.
- **get_all_reports()**: Devuelve todos los informes generados.
- **get_pre_static_analysis(sha256)**: Devuelve el análisis pre estático de la aplicación con ese sha256.
- **get_dynamic_analysis(sha256)**: Devuelve el análisis dinámico de la aplicación con ese sha256.
- **get_dynamic_analysis_droidbox(sha256)**: Devuelve sólo la parte de DroidBox del análisis dinámico de la aplicación con ese sha256.
- **get_dynamic_analysis_strace(sha256)**: Devuelve sólo la parte de Strace del análisis dinámico de la aplicación con ese sha256.
- **get_virusTotal_analysis(sha256)**: Devuelve el análisis de VirusTotal de la aplicación con ese sha256.
- **get_static_analysis(sha256)**: Devuelve el análisis estático de la aplicación con ese sha256.
- **get_static_analysis_androPyTool(sha256)**: Devuelve sólo la parte de AndroPyTool del análisis estático de la aplicación con ese sha256.
- **get_static_analysis_flowDroid(sha256)**: Devuelve sólo la parte de FlowDroid del análisis estático de la aplicación con ese sha256.
- **get_complete_analysis(sha256)**: Devuelve el análisis completo de la aplicación con ese sha256.

La API creada tiene los siguientes *endpoints*:

- **GET /reports**: Devuelve una lista con los informes que se han generado en formato json.
- **GET /reports/<sha256>**: Devuelve el análisis pre estático de la apk que se ha analizado (Filename, md5, sha1 y sha256). En caso de que la aplicación analizada no fuera un apk válido devuelve 200 OK con el mensaje “Apk is invalid“. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/complete**: Devuelve el informe completo de la apk que se ha ana-

lizado. En caso de que no exista devuelve 404 Not Found.

- **GET /reports/<sha256>/dynamic**: Devuelve sólo la parte dinámica del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/dynamic/droidbox**: Devuelve sólo la parte de DroidBox del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/dynamic/strace**: Devuelve sólo la parte de Strace del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/virustotal**: Devuelve sólo la parte de VirusTotal del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/static**: Devuelve sólo la parte estática del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/static/andropytool**: Devuelve sólo la parte de AndroPyTool del informe. En caso de que no exista devuelve 404 Not Found.
- **GET /reports/<sha256>/static/flowdroid**: Devuelve sólo la parte de FlowDroid del informe. En caso de que no exista devuelve 404 Not Found.
- **POST /files?virus_total_api_key=miApiKey**: Admite un solo fichero .apk de hasta 32MB y genera el informe usando androPyTool. Si la aplicación ya ha sido analizada manda directamente el uri de su informe. Si no lo ha sido, la analiza y devuelve su uri. Si especificas el parámetro *virus_total_api_key*, se ejecuta el análisis en VirusTotal usando la API key que le hayas especificado. Sino se usa por defecto la de androPyTool. En caso de que el archivo a analizar no sea una apk o intentes subir varios archivos, devuelve 400 Bad Request.

3.1.2. Desarrollo

Para el desarrollo de la API se ha creado una rama de Git en el proyecto de AndroPyTool donde se ha ido subido todo el código: https://github.com/alexMyG/AndroPyTool/tree/restApi.

Para la estructura de ficheros se ha mantenido en la medida de lo posible la estructura del proyecto original, para poder asegurar que al actualizar esta rama no se producen conflictos al hacer *merge*. Se han creado las siguientes archivos:

/
- **rest_api_app.py**: Punto de entrada de la API.
- **rest_api**
  - **files**: Carpeta donde se guardan los informes generados.
  - **virus_total_api_key**: Credenciales de VirusTotal a usar por defecto (No subidas a Git).
  - ***_controller.py**: implementación de la capa de la API.
  - ***_service.py**: implementación de la capa de lógica de negocio.
  - ***_repository.py**: implementación de la capa de acceso a datos.

Para implementar la arquitectura hexagonal se ha optado por dividir las diferentes lógicas de la aplicación en controladores, servicios y repositorios.

El módulo principal registra dos controladores, uno para recuperar informes y otro para analizar aplicaciones.

Los controladores implementan la interfaz de usuario. En nuestro caso se encargan de exponer la API, gestionar los *endpoints* y realizar la serialización y deserialización de datos. Además se encargan de enviar los códigos http de error correspondientes cuando se producen errores. Los controladores llaman a sus respectivos servicios para que se encarguen de la lógica de negocio.

El servicio de informes (*reports_service.py*) contacta con su repositorio para obtener los diferentes tipos de informes y enviarlos al controlador. El servicio de archivos llama a AndroPyTool para procesar un archivo y posteriormente llama al repositorio de archivos para guardar la aplicación y al repositorio de informes para guardar el informe generado en base de datos.

Los repositorios se encargan de gestionar la base de datos y recuperar de ella los archivos que se piden

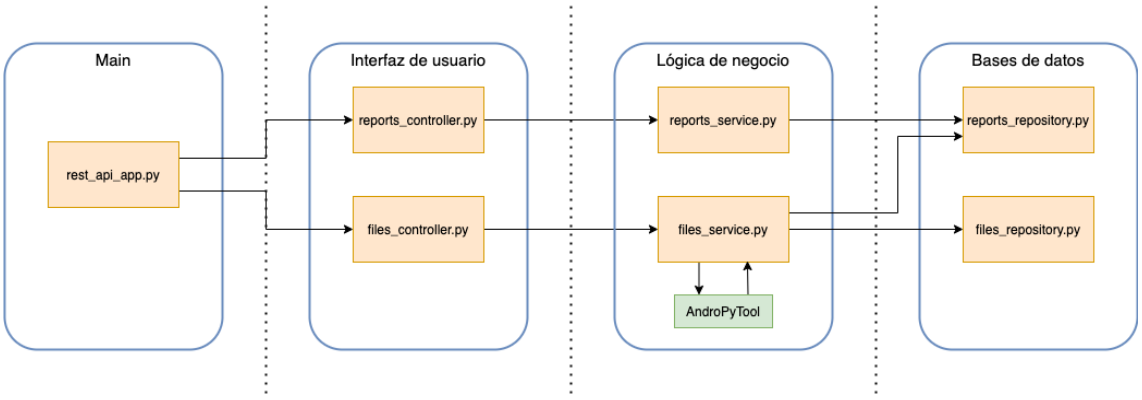


Figura 3.2: Diagrama de la lógica de llamadas seguida por la aplicación.

Como hay 2 *endpoints* fundamentales, /reports y /files, hay un controlador, servicio y repositorio específico para cada uno de ellos.

rest_api_app.py

En *rest_api_app.py* se inicializa Flask y se configuran los parámetros a usar:

- Se especifica el tamaño máximo de ficheros a subir.
- Se aplica la política de CORS.
- Se lee la API key de VirusTotal a usar por defecto.
- Se registran los 2 *endpoints* definidos en files_controller y reports_controller.

En caso de que sea la primera vez que se ejecuta la aplicación, se crea un fichero vacío que lista todas las aplicaciones analizadas. Es una mala práctica subir credenciales al versionado de código, pero la aplicación las necesita para ejecutar los análisis de VirusTotal. Por ello se comprueba si existe el archivo con las credenciales y, en caso de que no exista, se muestra un mensaje de error. Por último, en caso de que la aplicación se esté lanzando en el entorno de desarrollo, se lanza un servidor de pruebas.

Controladores

En *reports_controller.py* están definidos todos los *endpoints* que empiezan por /reports.

En *files_controller.py* están definidos todos los *endpoints* que empiezan por /files.

Se utiliza el decorador de Flask, `route()`, para definir qué función se ejecutará al llamar a cada uno de los *endpoints*. La mayoría de estas funciones simplemente llaman a la capa de negocio y devuelven ese resultado. Como los resultados son archivos json, Flask se encarga automáticamente de enviarlo de la manera adecuada.

Las únicas dos funciones que no son tan simples, son el controlador de /files y el controlador de /reports/<sha256>/dynamic/strace. Este último no devuelve un json sino un csv y tenemos que usar la función de Flask `send_file()`.

El controlador de /files primero comprueba que no se envíen múltiples archivos en una sola petición y que el archivo recibido es una apk. En caso de que no lo sea hemos creado la función auxiliar `throw_error(error_message, error_code)`, que manda un error con el código que le especifiques.

Por seguridad, todas las variables que puede especificar el usuario por url, como por ejemplo en /reports/<sha256>, primero se normalizan y luego se mandan a la capa de negocio para así evitar posibles ataques de XSS (*Cross Site Scripting*).

Servicios

files_service.py se encarga de la lógica de analizar una aplicación. Primero llama al repositorio para comprobar si la aplicación ya ha sido analizada previamente. Si ya lo ha sido, devuelve la uri del recurso con el código 200 OK. Si no lo ha sido, se ejecuta el análisis estático y dinámico de la aplicación usando `execute_andro_py_tool_steps()`, se guarda el informe y se devuelve la uri del recurso con el código 200 Accepted.

reports_service.py se encarga de la lógica de conseguir los diferentes informes de una aplicación. Primero llama al repositorio para comprobar si la aplicación ya ha sido analizada. Si no lo ha sido devuelve 404 Not Found. Si la aplicación sí ha sido analizada previamente, se vuelve a llamar al

repositorio para conseguir el informe.

Repositorios

reports_repository.py se encarga de recuperar los informes generados. Se ha optado por no usar una base de datos sino directamente usar el sistema de ficheros del sistema. Esta era la solución más rápida de implementar debido a un *bug* en AndroPyTool que no permite usar MongoDB.

Para cada una de las aplicaciones analizadas, nada más empezar el análisis, se crea una carpeta de nombre el *hash* del apk. Dentro de esta carpeta se guardan todos los archivos relacionados con el análisis de la aplicación.

Debido a esta lógica, para comprobar si una aplicación ha sido analizada o está siendo analizada, la función `app_has_report(sha256)` sólo necesita comprobar si existe una carpeta creada cuyo nombre sea el *hash* de la aplicación.

Al acabar el análisis de una aplicación, siempre se llama a `save_report_to_db(sha256)`, que modifica el informe generado para que tenga el nombre correcto de la apk, la fecha de análisis correcta y que el análisis de Strace tenga de link del *endpoint* en vez de la carpeta usada internamente. Además se guardan los datos del análisis pre estático de la aplicación en el fichero *all_reports.json*.

Las funciones `get_*(sha256)` leen el informe generado para la aplicación y devuelven la parte consultada. En caso de que no se encuentre el informe, se comprueba si la aplicación ha sido marcado como inválida. Si es inválida se devuelve 200 OK con el mensaje: “Apk is invalid”. En caso de que no se haya marcado como inválida, significa que la aplicación todavía está siendo analizada o que ha habido algún error y se devuelve 500 Internal Server Error.

Por temas legales no sabemos si podemos hacer pública la parte del análisis correspondiente a VirusTotal. Por ello, aunque sí que almacenemos esos datos, el repositorio filtra siempre los resultados de VirusTotal y los sustituye por un link al informe generado en su página. Si en un futuro sabemos que se pueden usar esos datos bastaría con eliminar ese filtro y se mandaría el informe completo.

3.2. Front end

3.2.1. Diseño

Para el *front end* hemos usado Angular y la librería de componentes de Material Design. Se han desarrollado principalmente dos elementos: componentes y servicios.

La función de los componentes es pintar en pantalla sin realizar lógicas complejas. Se encargan de la lógica de la interacción con el usuario y la lógica compleja la delegan en los servicios. Los componentes se declaran dentro de una carpeta compuesta por un 4 ficheros:

- un documento **.component.html*
- una hoja de estilos **.component.scss*
- la lógica en un archivo **.component.ts*
- los tests del componente en **.component.spec.ts*

Los servicios se encargan de lógicas más complejas como por ejemplo, las llamadas a la API. Se declaran dentro de una carpeta compuesta por 2 ficheros: un archivo **.service.ts* y sus tests en un archivo **.service.spec.ts*.

Al usar Angular, tenemos una página SPA en la que tenemos un elemento común a todas las pantallas. Es una barra de navegación en la parte superior con el logo de AndroPyTool y un selector de idioma, como se puede ver en la Figura 3.3.

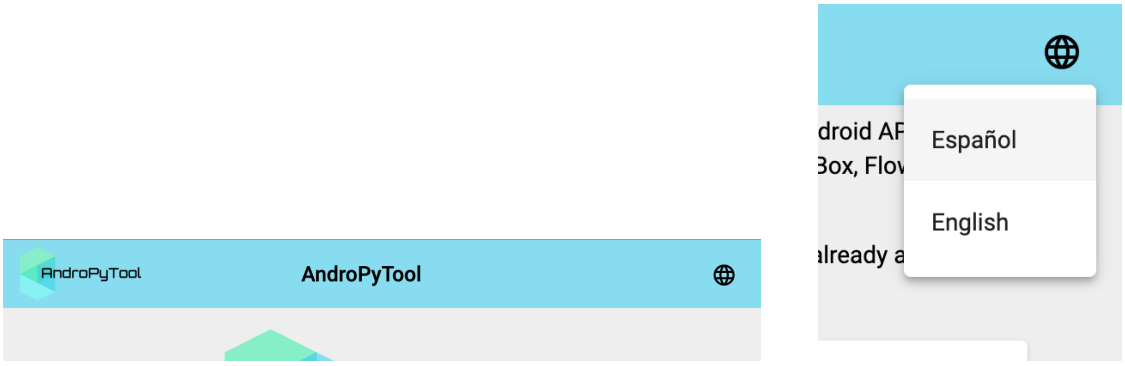
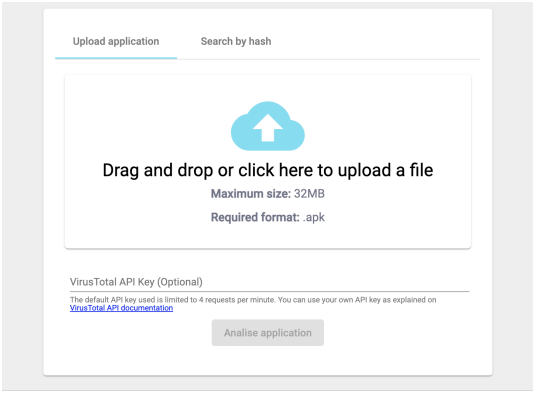


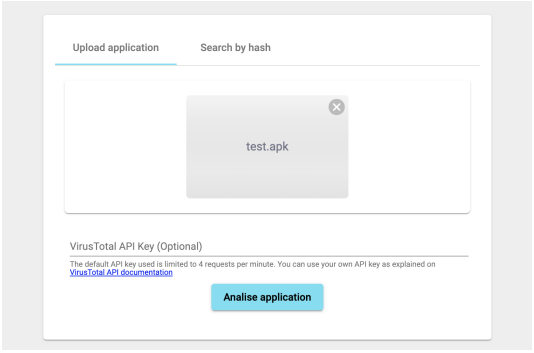
Figura 3.3: Barra de navegación y detalle del selector de idiomas.

En la página principal tenemos una caja en la que podemos subir archivos arrastrándolos o haciendo click en la caja. Una vez que hemos subido el archivo podemos borrarlo con un botón de “X” o pulsar en el botón de analizar aplicación. Nos aparecerá una pantalla de carga avisando que es un proceso largo y que se nos redirigirá al informe una vez acabado. Podemos ver estas pantallas en la

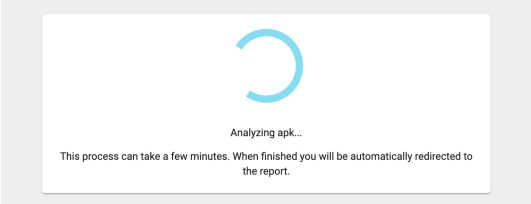
Figura 3.4. La página del informe podemos verla en la Figura 3.7.



(a) Estado inicial.



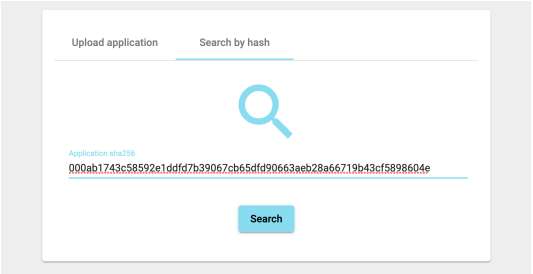
(b) Estado tras subir el archivo test.apk.



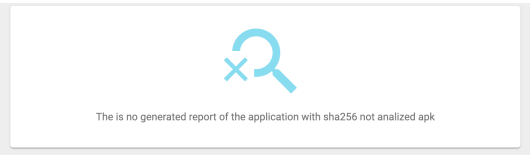
(c) Estado tras pulsar en el botón de analizar.

Figura 3.4: Pantalla principal.

En la página principal también podemos cambiar de pestaña y tenemos un buscador en el que podemos introducir el *hash* sha256 de una aplicación y se nos redirigirá a su informe. En caso de que no exista se nos redirigirá a una página de error. Podemos ver estas pantallas en la Figura 3.5.



(a) Búsqueda por hash.



(b) Página de error ya que no se ha analizado ninguna apk con *hash* "not analized apk".

Figura 3.5: Buscador por sha256.

Desde la página principal podemos navegar a la página de informes, en la que se nos listan todos los informes generados con unos pocos detalles de cada uno. Podemos ver esta pantalla en la Figura 3.6.

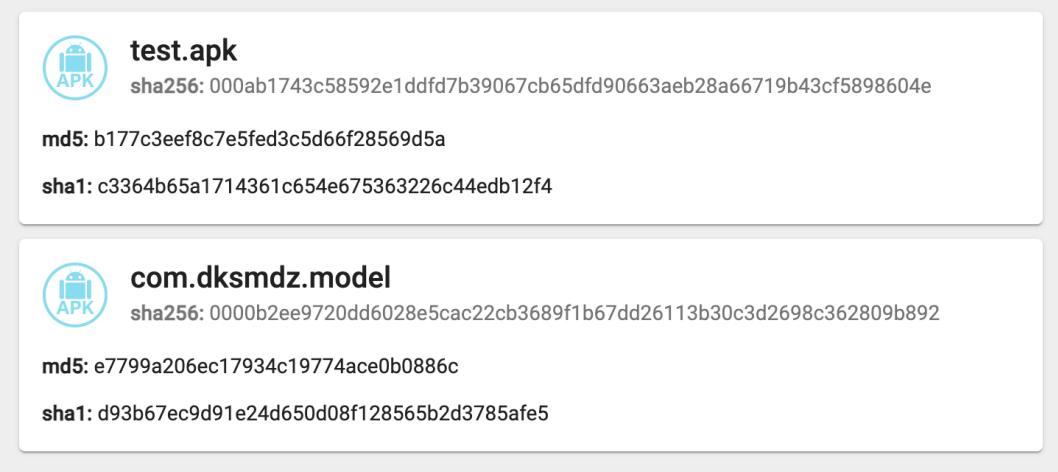
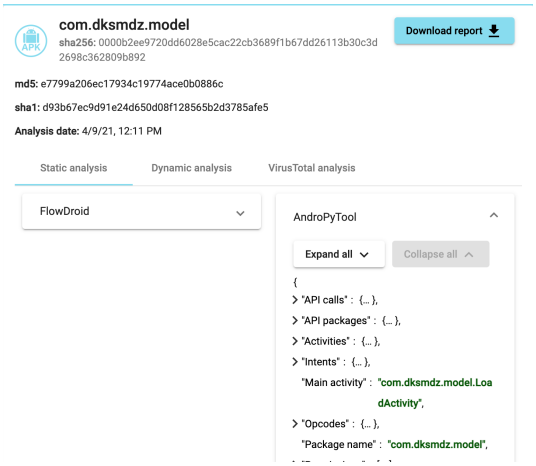


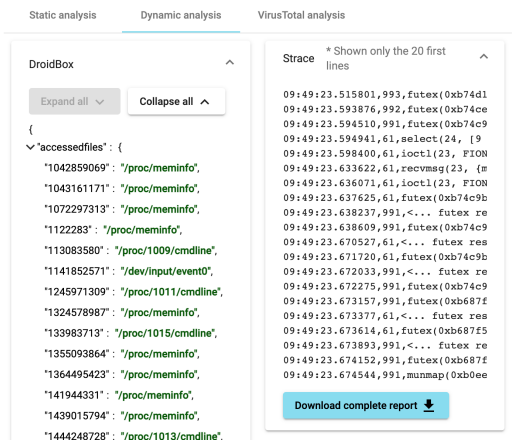
Figura 3.6: Lista de todas las aplicaciones analizadas.

Desde esta página podemos hacer click en cualquiera de los informes y se nos abrirá una pantalla con el análisis detallado. Esta es la misma pantalla a la que se nos redirige cuando subimos un archivo en la página principal o usamos el buscador.

En la pantalla del análisis, tenemos un botón arriba a la derecha que nos permite descargar el informe completo en formato json. Tenemos varias pestañas en las que podemos visualizar el análisis estático, dinámico y de VirusTotal. Dentro del análisis estático podemos ver los subanálisis de FlowDroid y de AndroPyTool. Dentro del análisis dinámico podemos ver los subanálisis de DroidBox y de Strace. Estos subanálisis se pueden expandir y contraer para comodidad del usuario. Los análisis en formato json se muestran en forma de árbol. Del análisis de Strace se muestran sólo las 20 primeras líneas y se muestra un botón para descargar el informe de Strace completo.



(a) Informe de una aplicación. El análisis de FlowDroid está contraído y el de AndroPyTool expandido. El informe de AndroPyTool se presenta en forma de árbol.



(b) Informe dinámico. Del análisis de Strace se muestran sólo las 20 primeras líneas y hay un botón para descargar el informe completo.

Figura 3.7: Análisis detallado.

3.2.2. Desarrollo

Para el desarrollo del portal web se ha creado un proyecto en GitHub donde se ha ido subido todo el código: https://github.com/ArturoBlazquez/AndroPyTool-angular.

La estructura de ficheros que se ha seguido es la siguiente:

/
- *: Archivos de configuración.
- **src**
  - **environments**: Archivos de configuración de los entornos de producción y desarrollo.
  - **assets**: Recursos como imágenes y librerías.
    - **i18n**: Archivos de traducción.
  - **app**
    - **models**: Modelos de datos.
    - **pages**
      - **upload**: Componente que pinta la pantalla principal.
      - **reports**: Componente que pinta la lista de informes.
      - **report-detail**: Componente que pinta un informe detallado.
    - **services**: Servicios usados por los componentes
    - **shared**: Componentes que se reutilizan en varias páginas

package.json

En este archivo archivo se definen las dependencias del proyecto, que podrán ser instaladas ejecutando `npm install`.

Se definen también una serie de comandos útiles como `npm run start`, que levanta un servidor de desarrollo o `npm run build`, que compila la aplicación.

Entornos

Hemos definido dos entornos, el de desarrollo y el de producción. La diferencia entre ellos es que en desarrollo se llama a la api levantada en local y en producción se llama a la api levantada en el servidor de producción.

Traducción

La página se puede usar en inglés y en español.

Para conseguir esto hemos usado la librería ngx-translate y hemos usado cadenas en vez de escribir los textos directamente en el html o en el código. En la carpeta i18n hay dos archivos llamados *en.json* y *es.json* que contienen las traducciones de las cadenas a inglés y español, respectivamente.

La librería se encarga de encontrar la traducción de las cadenas y en función del idioma que hayamos elegido se muestra en un idioma o en otro.

En el componente principal de la aplicación registramos las traducciones a español e inglés y elegimos por defecto el idioma del navegador del usuario con el siguiente código:

```
constructor(
    private readonly translate: TranslateService,
    ...
) {
    translate.addLangs(['en', 'es']);

    const browserLang = translate.getBrowserLang();
    if (browserLang !== 'en' && browserLang !== 'es') {
      translate.setDefaultLang('es');
    } else {
      translate.setDefaultLang(browserLang);
    }
}
```

Una vez registradas las traducciones, usarlas en html es tan sencillo como hacer:

```
<p>{{'text.to.translate' | translate}}</p>
```

y en cada uno de los ficheros de traducción añadir la traducción de la cadena:

```
{
    "text.to.translate": "Texto traducido"
}
```

Para usar las traducciones en un archivo **.ts* primero tenemos que inyectar el servicio de traducción en el constructor del componente y luego lo podemos llamar usando la función instant:

```
constructor(
    private readonly translate: TranslateService,
    ...
) {
    ...
}

this.translate.instant('text.to.translate')
```

Servicios

Hemos definido 3 servicios, FileService, ReportService y TitleService.

FileService y ReportService se encargan de hacer las llamadas a la api y transformar estas llamadas a los objetos que usa nuestra aplicación. Para ello hemos usado la librería rxjs que simplifica mucho la lógica de las llamadas.

Por ejemplo, para llamar al *endpoint* que nos devuelve todos los informes, nos basta con hacer esto:

```
this.httpClient.get<ReportsResponse>(environment.restApiUrl + 'reports')
```

En vez de poner directamente la url de la api, la tomamos del archivo de configuración de entornos y así podemos tener la misma lógica en desarrollo y en producción. Una vez hecha la llamada podemos usar la función `pipe` de rxjs para procesar la respuesta o gestionar errores.

TitleService es un servicio que cambia el título de la página. Este servicio traduce automáticamente el texto a mostrar.

Componentes comunes

Hemos definido 2 componentes que se reusan en diferentes lugares de la aplicación.

DownloadButtonsComponent pinta el botón de descargar el informe completo. Para usarlo necesitas especificar el id del informe. Al hacer click en el botón llama al servicio reportService y guarda el fichero en descargas.

JsonTreeComponent pide como argumento unos datos en formato json. Se encarga de pintar los datos en formato de árbol, permitiendo plegar y desplegar las diferentes partes del archivo json. Pinta también dos botones que permiten expandir o contraer todos los nodos del árbol.

Rutas

Dependiendo de la ruta a la que el usuario haya navegado, hemos especificado a Angular que se encargue de pintar diferentes componentes.

- Si navegas a http://<AndroPyToolUrl>/ se pinta el componente UploadComponent
- Si navegas a http://<AndroPyToolUrl>/reports se pinta el componente ReportsComponent
- Si navegas a http://<AndroPyToolUrl>/reports/:reportId se pinta el componente ReportDetailComponent
- En caso de que se navegue a cualquier otra ruta se pinta el componente UploadComponent

Cada uno de estos componentes usa el servicio TitleService para actualizar el título de la página. Por ejemplo, si estás en la página de un informe, el título es “AndroPyTool - Informe” si tienes la página en español y “AndroPyTool - Report” si tienes la página en inglés.

UploadComponent

Es el componente que se pinta como página principal. Usamos el componente Tabs de Material Design para pintar dos pestañas, una que permite subir archivos y otra que es un buscador.

Para subir archivos hemos usado el componente ngx-dropzone que permite subir archivos tanto arrastrando como haciendo click en la caja. Lo hemos configurado para que sólo acepte un archivo con extensión .apk y de menos de 32MB

El componente pinta también el logo de AndroPyTool con una descripción de qué es la página y cómo usarla y un link a todos los informes generados.

Para analizar el archivo subido usamos el servicio FileService y al acabar el análisis se redirige a la página del informe. En caso de la api haya devuelto un error (por ejemplo si el servidor de *back* no está levantado) se muestra un popup con el mensaje de error recibido.

ReportsComponent

Este componente pinta una lista con una breve información de cada uno de las aplicaciones analizadas. Para ello usa el servicio ReportService para obtener todos los informes generados y los pinta usando el componente Card de Material Design.

ReportDetailComponent

Este componente pinta un informe detallado. En la parte superior del informe se pintan los datos generales del informe: el nombre de la aplicación, sus *hashes* y la fecha en la que fue analizada. Utiliza el componente común DownloadButtonsComponent para permitir descargar el informe completo.

En la parte inferior utiliza el componente Tabs para pintar 3 pestañas que contienen los análisis estático, dinámico y de VirusTotal.

Los datos de cada una de estas pestañas se consiguen usando el servicio ReportService. Usamos el componente Expansion Panel de Material Design para pintar los datos recibidos. Si los datos son en formato json usamos el componente común JsonTreeComponent para pintar los datos dentro de su correspondiente Expansion Panel.

Para los datos de Strace pintamos sólo las 20 primeras líneas y hay un botón que permite descargar el informe completo. Como el informe de Strace puede llegar a ser muy pesado, no lo pintamos nada más cargar la página, sino que se hace al navegar a la pestaña de análisis dinámico.

Este componente también se encarga de pintar diferentes pantallas en caso de que el análisis no sea un análisis estándar. Si el análisis no existe, se pinta la página que se puede ver en la Figura 3.5. Si el análisis es de una aplicación inválida, se muestra la pantalla que se puede ver en la Figura 3.8:

Si el análisis está siendo generado en estos momentos, se muestra la pantalla de carga que puedese ver en la Figura 3.9:

Accesibilidad

Este portal web ha sido diseñado teniendo en cuenta la accesibilidad como un aspecto fundamental y por lo tanto es completamente operable por teclado y cumple las normativas de accesibilidad WCAG 2.1.

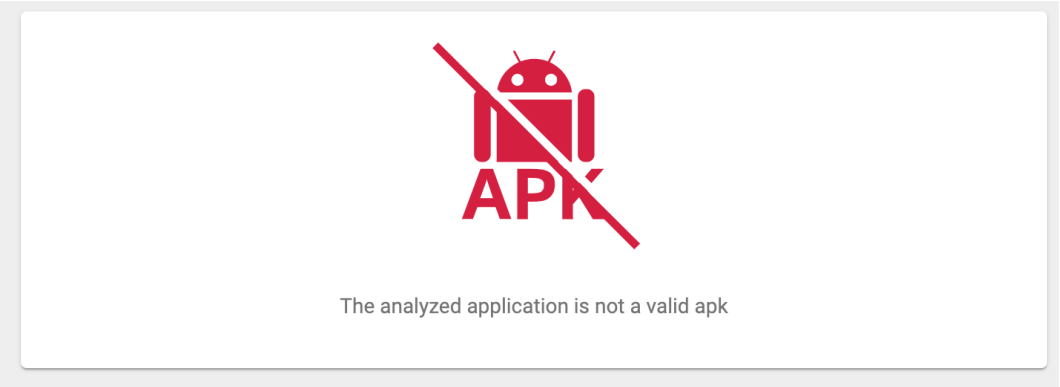


Figura 3.8: Pantalla de informe para una aplicación inválida.

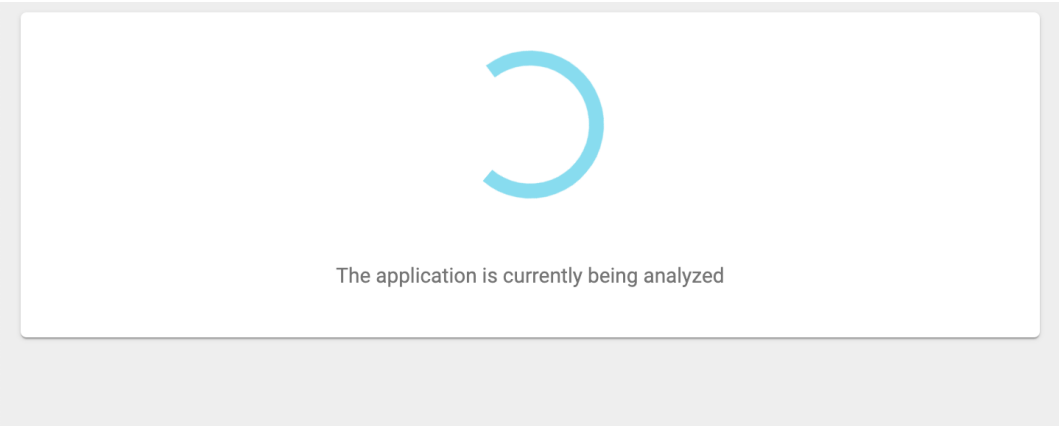


Figura 3.9: Pantalla de informe mientras se genera el análisis.

Responsive

La página es completamente adaptable a diferentes tamaños de pantallas, desde los ordenadores más grandes hasta los móviles con menor resolución. La página está diseñada de manera que da igual el ancho del navegador, la experiencia de navegación es siempre la misma.

Además, en caso de que estemos usando un móvil para navegar por la página, hemos añadido en la barra de navegación un botón de retroceder a la página anterior para mejorar la experiencia de usuario.

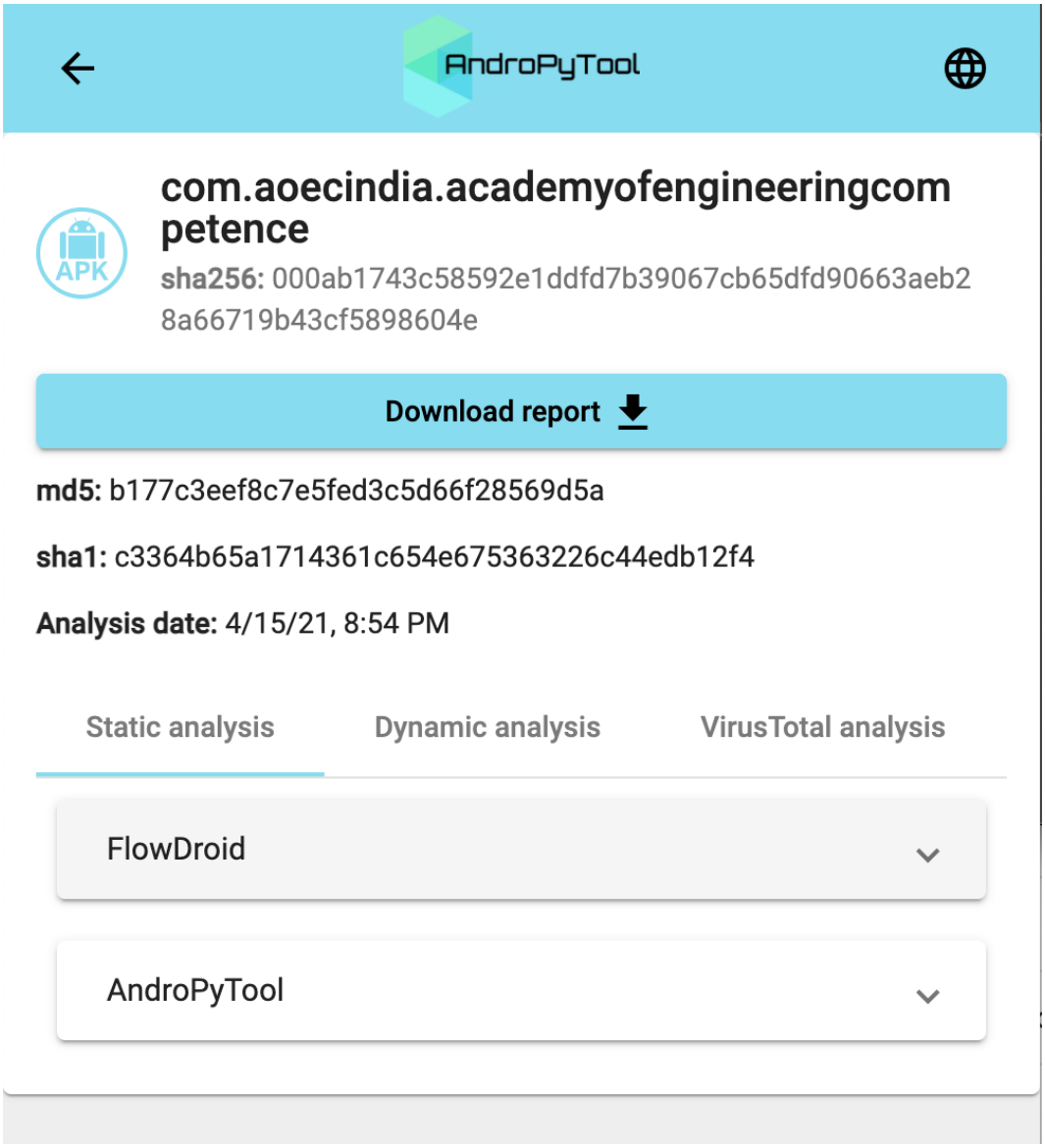


Figura 3.10: Pantalla de informe en móviles. Se puede ver arriba a la izquierda una flecha para restroceder a la pantalla anterior

3.2.3. Integración front y back

El portal web está dividido en componentes, que pintan la interfaz que ven los usuarios. Muchos de estos componentes sólo dibujan en la pantalla y no necesitan interactuar con el *back end*. Los componentes que sí necesitan interactuar con el *back end* lo hacen a través de los servicios definidos en el *front end*. Estos servicios se encargan de hacer la correspondiente petición http al *back end* y procesar la respuesta para controlar posibles errores y transformar los datos recibidos en las clases de datos que utiliza el portal web.

La integración del *front end* y *back end* se hace mediante llamadas *http* a la API. Así se logra que ambas capas estén muy desacopladas y su integración es muy sencilla ya que estamos usando un protocolo muy estandarizado y robusto.

En la figura 3.11 puede verse un esquema de una pantalla del portal web. Tenemos múltiples componentes que pintan las diferentes partes de la página web. El *Componente 2* pinta datos que hay que obtener del *back end*. Para ello delega la responsabilidad de contactar con el *back end* en el *Servicio 1*, que envía una petición http al *endpoint* de la API correspondiente. El *back end* responde con un archivo json, que el *Servicio 1* se encarga de deserializar para transformarlo en la clase de datos que utiliza el *Componente 2*.

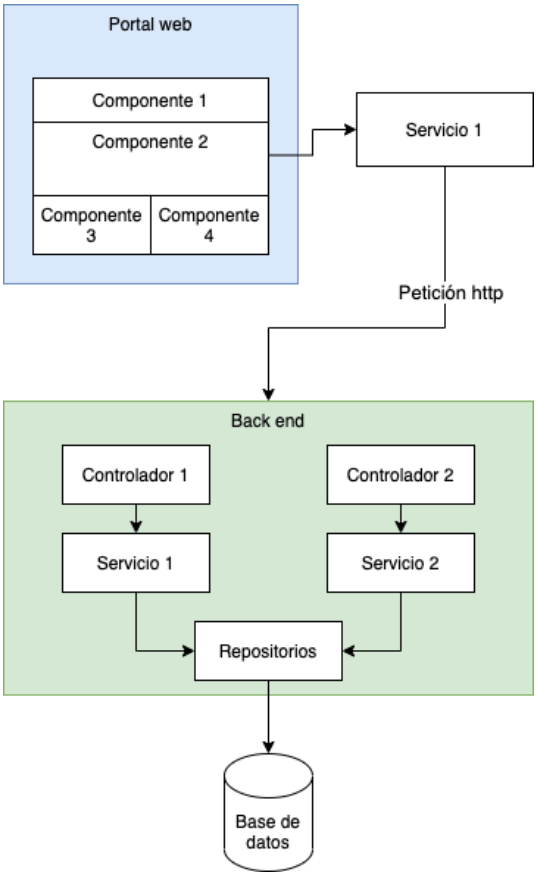


Figura 3.11: Diagrama de la integración *front* y *back*.

3.3. DevOps

Para el desarrollo en local se han configurado entornos de desarrollo tanto para *back* como para *front*.

Durante las primeras etapas del desarrollo de la api en *back*, la aplicación se ejecutaba directamente desde la consola del IDE, usando el servidor de desarrollo que proporciona Flask. Para comprobar que la api generada funciona correctamente se ha usado Postman. En la Figura 3.12 podemos ver todos los tests definidos, que comprueban que cada uno de los *endpoints* definidos funcionan correctamente. Gracias a las comodidades que nos ofrece Postman, hemos usado esta misma colección de peticiones para comprobar que la api en producción también funciona correctamente. Con estas peticiones, además de comprobar que las peticiones normales funcionan correctamente, hay una carpeta con peticiones que deberían dar error para comprobar que la gestión de errores de la aplicación también es correcta.

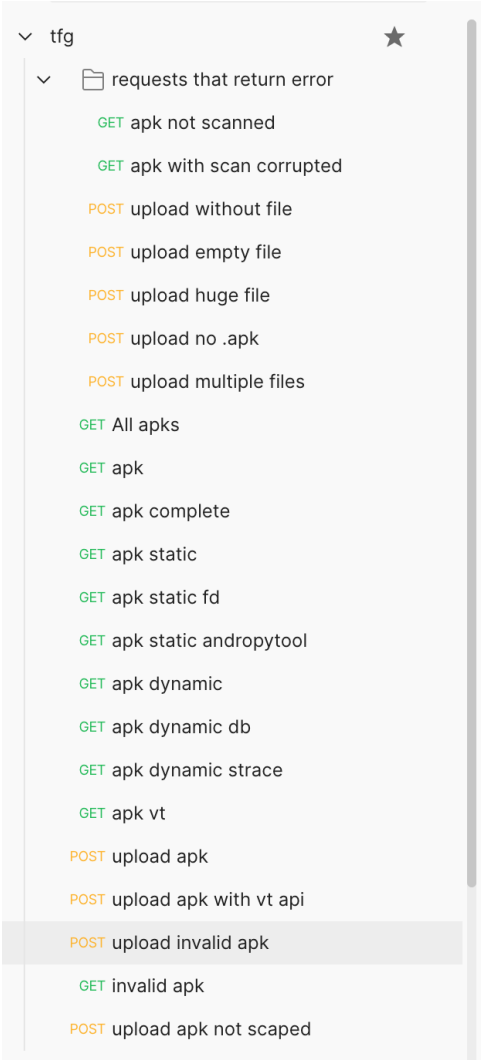


Figura 3.12: Colección de peticiones de Postman.

Una vez que el desarrollo de la api se volvió más estable, se creó un archivo Dockerfile dentro del proyecto que levanta un servidor de Gunicorn, que es un servidor pensado para entornos de producción. Hemos configurado el servidor para que levante 20 *workers* y no dé *timeout* en las peticiones a no ser que duren más de 10 minutos. Hemos puesto este *timeout* tan alto ya que el análisis de las aplicaciones puede llegar a durar más de 5 minutos.

De esta manera, usando Docker, podemos levantar en nuestro local el mismo servidor que se va a levantar en producción y hacer unas pruebas más realistas. Al levantar el contenedor especificamos el puerto en el que queremos que escuche la api. Como los contenedores usan su propio sistema de archivos, que es borrado una vez que el contenedor es detenido, hemos configurados un volumen para persistir en nuestro sistema los informes de las aplicaciones analizadas.

Para el desarrollo en *front* hemos configurado el archivo de entornos para que llame a la api que tenemos levantada en docker en nuestro local. Durante el desarrollo se ha levantado un servidor en local ejecutando el comando `ng serve`, que levanta un servidor con la página y que se actualiza automáticamente cuando haces cambios en el código. Una vez terminado el desarrollo del *front* y tras haber levantado la api en producción, cambiamos el fichero de entornos para que apuntara a la api en producción y hacer pruebas más realistas antes de subir el *front* a producción.

Para desplegar las aplicaciones en producción lo único que se necesita es una cuenta con permisos de Docker en el servidor de producción. Con esa cuenta se puede acceder al servidor por ssh y usar git para descargarse el código del *back* y del *front*. A continuación es necesario copiar los archivos de configuración y ficheros sensibles que no se han subido a GitHub. En nuestro caso tenemos que copiar en la carpeta de *back end* el archivo `rest_api/virus_total_api_key`.

Para desplegar de manera más cómoda hemos creado un Dockerfile que levanta un servidor Apache2, compila la aplicación y sirve el portal web. También se ha creado un *script* `deploy.sh` que automatiza el proceso de desplegar tanto el *back* como el *front*. Este script comprueba si hay contenedores levantados del *front* o del *back* y los detiene. Después se baja los nuevos cambios que haya habido en los repositorios, vuelve a crear las imágenes de Docker y levanta los contenedores con el código actualizado. Se le puede llamar con la bandera `-only-front`, que hace lo mismo pero sólo para el contenedor del *front*. También se puede llamar con la bandera `-delete-images`, que además de detener los contenedores borra sus imágenes asociadas.

4

Conclusiones

4.1. Conclusiones y lecciones aprendidas

En este proyecto se ha implementado una aplicación web que permite a los analistas de *malware* analizar aplicaciones de Android de una manera muy sencilla. Además esta plataforma permite descargar y compartir los informes que se han generado sobre ellas, permitiendo así una colaboración efectiva entre diferentes investigadores o diferentes equipos de investigadores.

La aplicación cumple con todos los requisitos definidos y el resultado final es un portal web con una experiencia de usuario muy agradable, completamente *responsive* y que cumple todas las normativas de accesibilidad.

En este trabajo se han llevado a cabo todos los aspectos de un proyecto, desde la definición de requisitos, pasando por el diseño y desarrollo de la aplicación hasta llegar al despliegue y gestión del ciclo de vida de las aplicaciones ya desarrolladas. Además se ha pensado en el mantenimiento de la aplicación y posibles desarrollos futuros.

La arquitectura usada en el *back* permite cambiar completamente las diferentes capas de la aplicación, de manera que cuando haya actualizaciones de la herramienta de AndroPyTool no habrá que hacer ningún cambio en las capas que no son de negocio y en la capa de negocio bastará con hacer un `git pull`.

La arquitectura usada para el *front* permite una rápida comprensión del código, facilitando poder añadir funcionalidades nuevas y el mantenimiento futuro.

El despliegue automatizado de las aplicaciones permite que se puedan hacer desarrollos futuros tanto de la parte del *front* como la del *back* y que los desarrolladores no tengan que preocuparse por la parte de DevOps, sino simplemente de desarrollar y programar.

Gracias a este trabajo he aprendido nuevos lenguajes, herramientas y servicios, afianzando así mis conocimientos como desarrollador. He podido poner en práctica todos los conocimientos aprendidos de Ingeniería de Software en un problema del mundo real.

4.2. Trabajo futuro

Esta aplicación es completamente funcional pero, como todo, puede ser mejorada. La mayoría de mejoras pueden ser hechas en la parte del *back end*.

Actualmente se utiliza como gestión de datos una estructura de ficheros en vez de una base de datos. Esto no supone un problema actualmente, puesto que el principal cuello de botella está en el análisis de las aplicaciones, pero lo preferible sería usar una base de datos. Para ello habría que solucionar un *bug* que hay en la herramienta de AndroPyTool al usar MongoDB. Al haber usado una arquitectura hexagonal este cambio implicaría cambiar únicamente la parte de acceso a datos, sin tener que tocar nada de ninguna de las demás capas.

Hay un *bug* que hace que si una aplicación está siendo analizada y a la vez se hace una petición para analizar otra, puede hacer que el análisis de ambas falle. Esto es debido al análisis dinámico de DroidBox, que no permite correr 2 instancias a la vez. Una posible solución sería tener un balanceador de carga que levantara un contenedor de Docker por cada análisis a ejecutar.

Otro posible trabajo futuro sería trabajar en la seguridad de la api. Actualmente no está protegida frente a posibles ataques DDoS y habría que limitar el número de análisis que puede pedir un único usuario para evitar que una sola persona bloquee todos los recursos de la máquina.

También, debido a que el análisis es un proceso muy largo (puede llegar a durar más de 5 minutos), sería muy útil para el usuario conocer más detalles de en qué estado se encuentra actualmente el análisis y tener una barra de progreso.

De momento no es un problema, puesto que no se han generado muchos informes, pero cuando haya miles, la petición para recuperar todos los informes debería estar paginada.

Como último punto para un posible trabajo futuro estaría comprar los dominios www.andropytool.com y www.andropytool.es para alojar ahí el portal web y no tener que acceder desde una dirección IP.

Bibliografía

[1] Martín, A., Lara-Cabrera, R., & Camacho, D.: *Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset.*. Information Fusion. DOI: 10.1016/j.inffus.2018.12.006 (2018).

[2] Martín, A., Lara-Cabrera, R., & Camacho, D.: *A new tool for static and dynamic Android malware analysis*. In Data Science and Knowledge Engineering for Sensing Decision Support (pp. 509-516). World Scientific. (2018).

[3] Malwarebytes Labs: *2020 State of Malware Report*. Online: https://resources.malwarebytes.com/resource/state-of-malware-report-2020/ Visitada el 5 de Junio de 2021. (2020).

[4] Androguard: Online: https://github.com/androguard/androguard Visitada el 5 de Junio de 2021. (2021).

[5] DroidBox: Online: https://github.com/pjlantz/droidbox Visitada el 5 de Junio de 2021. (2021).

[6] Apktool: Online: https://ibotpeaches.github.io/Apktool/ Visitada el 5 de Junio de 2021. (2018).

[7] dex2jar: Online: https://github.com/pxb1988/dex2jar Visitada el 5 de Junio de 2021. (2018).

[8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A.,Klein, J., et al.: *Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.*. O'Boyle, M.F.P., Pingali, K., editors. ACM SIGPLAN Con-ference on Programming Language Design and Implementation,PLDI '14. ACM, p. 259–269 (2014).

[9] OCTO Technology: *Hexagonal Architecture: three principles and an implementation*. Online: https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/ Visitada el 5 de Junio de 2021. (2018).

[10] Pallets: *Flask documentation*. Online: https://flask.palletsprojects.com/en/1.1.x/ Visitada el 5 de Junio de 2021. (2010).

[11] Benoit Chesneau: *Gunicorn documentation*. Online: https://docs.gunicorn.org/en/stable/settings.html Visitada el 5 de Junio de 2021. (2021).

[12] Docker Inc.: *Docker overview*. Online: https://docs.docker.com/get-started/overview/ Visitada el 5 de Junio de 2021. (2021).

[13] Google: *Introduction to the Angular Docs*. Online: https://angular.io/docs/ Visitada el 5 de Junio de 2021. (2021).

[14] Google: *Material Design*. Online: https://material.io/components/ Visitada el 5 de Junio de 2021. (2021).


UAM
UNIVERSIDAD AUTONOMA
DE MADRID